ChatGPT

SBERT (Sentence Transformers) Overview

Sentence Transformers (SBERT) is a popular library for computing dense and sparse text embeddings. It provides **SentenceTransformer** (bi-encoder) models for fast semantic similarity and search, **CrossEncoder** models for pairwise scoring (e.g. re-ranking), and **SparseEncoder** models (e.g. SPLADE) for sparse representations. SBERT includes tools for training, evaluating (e.g. MTEB), and optimizing models (quantization, ONNX). The following sections detail each component, usage patterns, models, training setup, evaluation, and best practices.

Sentence Transformer

Usage

A **SentenceTransformer** model encodes each input sentence independently into a fixed-size embedding. This is a bi-encoder architecture (often Transformer + pooling) enabling fast similarity or search (embeddings can be precomputed). Usage is simple: instantiate a model by name or path and call `.encode()`. For example:

```
from sentence_transformers import SentenceTransformer

# Load a pre-trained model (bi-encoder)
model = SentenceTransformer("all-MiniLM-L6-v2")
sentences = ["This is an example sentence", "Each sentence is encoded
separately"]
embeddings = model.encode(sentences)  # list of 384-d float32 vectors
```

By default SBERT will use the best available device (CUDA, MPS, or CPU). Models accept a variety of arguments (e.g. `model_kwargs={"torch_dtype":"float16"}`) to control precision and device [1]. Internally, most bi-encoder models are built from a **Transformer** module (e.g. BERT, RoBERTa) followed by a **Pooling** module that aggregates token representations [2]. For quick setup, simply provide a Hugging Face model name (e.g. `"bert-base-uncased"`).

Example: Compute cosine similarity between sentence embeddings:

```
from sentence_transformers import SentenceTransformer, util

model = SentenceTransformer("all-MiniLM-L6-v2")
emb1 = model.encode("Sentence 1")
emb2 = model.encode("Sentence 2")
```

```
sim = util.cos_sim(emb1, emb2)  # cosine similarity
print(f"Similarity: {sim.item():.4f}")
```

Internally, `SentenceTransformer` wraps a list of modules (Transformer, Pooling, etc.) for flexibility [3]. You can customize the modules (e.g. use a **StaticEmbedding** or add additional layers) by constructing with `modules=[...]`.

Pre-trained Models

SBERT offers a rich **model zoo** of pre-trained bi-encoder models on Hugging Face. These include models optimized for different tasks and domains:

- **Original Models:** Baseline SBERT models (e.g. `bert-base-nli-mean-tokens`).
- **Semantic Search / QA Models:** Models fine-tuned on MS MARCO or QA data (e.g. `msmarco-MiniLM-L12-v2`).
- **Multilingual Models:** Models trained for cross-lingual similarity (e.g. `paraphrase-xlm-r-multilingual-v1`) or bitext mining.
- **Domain-Specific:** Scientific or specialized corpora (e.g. *SciBERT*-based, or `Sentence-Transformer/instructor` models).

You can browse tagged Sentence Transformers models on Hugging Face with the `sentence-transformers` tag [4]. For example, many semantic similarity tasks use **MiniLM** or **DistilBERT** variants for efficiency, such as `"all-MiniLM-L6-v2"` or multilingual `"paraphrase-multilingual-MiniLM-L12-v2"`. Always choose base models well-aligned with your language or domain (e.g. multilingual XLM-R for non-English) [5].

Training Overview

SBERT training follows a **Trainer** paradigm (based on PyTorch and HuggingFace Datasets). Key components include:

- **Model:** Typically a Transformer+Pooling architecture. You can load an existing bi-encoder (e.g. `SentenceTransformer("bert-base-uncased")`) or build from modules [2].
- **Dataset:** A Hugging Face `Dataset` or `DatasetDict` with the appropriate format (see below). SBERT provides utilities (`ParallelSentencesDataset`, `SentenceLabelDataset`, etc.) and can load from HF Hub.
- **Loss Function:** Select a loss matching your data and task (MultipleNegativesRanking, Softmax, CosineSimilarity, etc.). SBERT has many loss classes (see *Loss Overview*).
- **TrainingArguments:** Hyperparameters (epochs, batch size, LR, etc.).
- **Trainer:** `SentenceTransformerTrainer` handles training loops, evaluation, checkpointing.

Example training code (fine-tuning on paired data):

```
from sentence_transformers import SentenceTransformer,
SentenceTransformerTrainer, losses
from datasets import Dataset
```

```
# Prepare model and data
model = SentenceTransformer("all-MiniLM-L6-v2")
data = [{"sentence1":"Hello", "sentence2":"Hi", "label":1.0}, ...]
train_dataset = Dataset.from_list(data)

# Choose a loss, e.g. cosine similarity loss for labeled pairs
loss = losses.CosineSimilarityLoss(model)

trainer = SentenceTransformerTrainer(
    model=model,
    train_dataset=train_dataset,
    loss=loss,
    # other args like train_batch_size, num_epochs...
)
trainer.train()
```

In practice, bi-encoder models often use **MultipleNegativesRankingLoss** (in-batch negatives) for unsupervised pairs, or **SoftmaxLoss/CrossEntropyLoss** for labeled data [6] [7]. A special **MSELoss** can distill knowledge from a teacher model (e.g. cross-lingual distillation) [8].

Training outputs learned sentence embeddings tailored to your notion of similarity. For example, a model fine-tuned on duplicate questions will embed paraphrases closer together, while a topic classifier fine-tune might cluster topical similarity instead.

Dataset Overview

Your dataset format must match the loss function. SBERT identifies four common formats [9]:

- **Positive Pairs:** Two related sentences per example (e.g. paraphrase pairs, (query, response), translation pairs). No explicit label needed.
- **Triplets:** (anchor, positive, negative) without labels (suitable for ranking losses).
- **Pairs with Score:** Two sentences + a float similarity score (0–1), typical for Semantic Textual Similarity (STS) tasks.
- **Class-labeled:** A single text + a class label. This can be converted to triplets by sampling positives/negatives from classes.

For example, the STS Benchmark (`sentence1, sentence2, score`) works with *CosineSimilarityLoss*, while a QnA dataset could use *MultipleNegativesRankingLoss* on (query, positive answer). SBERT's [Dataset Overview] suggests converting between formats when needed and warns that *dataset format must match the chosen loss* [10].

SBERT integrates Hugging Face Datasets:

```
from datasets import load_dataset
# Load a tagged Sentence Transformers dataset (see HF Hub)
```

```
dataset = load_dataset("sentence-transformers/stsb", split="train")
print(dataset.column_names)  # e.g. ['sentence1','sentence2','score']
```

Useful tips: clean up extra columns with `remove_columns`; convert class labels to triplets if necessary. For hard negatives, SBERT provides `sentence_transformers.util.mine_hard_negatives()` to create challenging (anchor, pos, neg) samples [11].

Loss Overview

Choosing the right loss is crucial. SBERT supports many losses. A **Loss Overview** table matches data formats to losses [12]. Highlights:

- **MultipleNegativesRankingLoss (MNR):** Popular for (anchor, positive) pairs without labels (InfoNCE, in-batch negatives). It is fast and effective [13].
- **CosineSimilarityLoss / CoSENT / AnglELoss:** Used for (sentenceA, sentenceB, score) STS tasks. CosineSimilarity is traditional, while CoSENT/AnglELoss often yield better performance [13].
- **ContrastiveLoss:** Uses (anchor, positive/negative, label) format (1/0) for training (supervised binary labels).
- **TripletLoss:** For (anchor, pos, neg) triplets, often less used than MNR for efficiency.
- **SoftmaxLoss (CrossEntropy):** For classification tasks (e.g. NLI) with class labels.
- **Distillation Losses (MSELoss, MarginMSELoss):** For distilling teacher similarity (e.g. cross-lingual distillation, model compression) [14].

SBERT also has **loss modifiers** like *MatryoshkaLoss* (for trainable embedding size) or *AdaptiveLayerLoss* (train layers to drop) [15]. Distillation losses like *MarginMSELoss* let you match teacher embeddings on pairs/triplets [14].

In practice, **MNR + (CachedMNR)** is often used for unsupervised pair data, and **Cosine/CoSENT** for STS. Custom losses can be implemented by subclassing `nn.Module` and following SBERT's API [16].

Training Examples

SBERT provides example scripts for common tasks:

- **Semantic Textual Similarity (STS):** Fine-tune on sentence pairs with a floating score using *CosineSimilarityLoss* (or CoSENT).
- **Natural Language Inference (NLI):** Use sentence pairs with entailment/contradiction labels; often *SoftmaxLoss* (3 classes) or *MultipleNegativesRankingLoss* on positive pairs.
- **Paraphrase Mining / Duplicate QA:** Fine-tune on question pairs (e.g. Quora, QQP) using *MultipleNegativesRankingLoss*.
- **MS MARCO (Bi-encoder):** Train bi-encoder retrieval model with (query, positive passages, in-batch negatives).
- **Matryoshka Embeddings / Adaptive Layers:** Advanced examples to train models that remain robust when pruned or truncated.
- **Multilingual Training:** Distillation with parallel corpora (see below).

For instance, training on STS could look like:

```
from sentence_transformers import SentenceTransformer,
SentenceTransformerTrainer, losses
from datasets import load_dataset

model = SentenceTransformer("all-MiniLM-L6-v2")
dataset = load_dataset("stsb_multi_mt", "en", split="train")  # example STS
dataset
dataset = dataset.map(lambda e: {'sentence1': e['sentence1'], 'sentence2':
e['sentence2'],
                                 'score': e['score']/5.0})
loss = losses.CosineSimilarityLoss(model)
trainer = SentenceTransformerTrainer(model=model, train_dataset=dataset,
loss=loss)
trainer.train()
```

A full suite of examples is available in the SBERT docs (see *Training Examples* section), covering each of the above scenarios with code snippets and data preparation tips.

Cross Encoder

Usage

A **CrossEncoder** jointly encodes a pair of sentences (or texts) and outputs a score or class label, typically by feeding the concatenated pair into a Transformer. Unlike bi-encoders, CrossEncoders do **not** produce independent sentence embeddings; instead they model interactions between the two inputs. This makes them **more accurate** for tasks like re-ranking or pairwise classification, but **much slower** (no precomputation).

Usage example (huggingface cross-encoder):

```
from sentence_transformers import CrossEncoder

# Load a pretrained cross-encoder (e.g. for MS MARCO ranking)
model = CrossEncoder("cross-encoder/ms-marco-MiniLM-L-6-v2")
pairs = [
    ("How many people live in Paris?", "The population of Paris is 2,148,000."),
    ("How many people live in Paris?", "Paris is famous for art museums."),
]
scores = model.predict(pairs)
print(scores)  # e.g. [8.60, -4.32] (higher = more relevant)
```

Or use it to **re-rank** candidates:

```
query = "How many people live in Berlin?"
passages = ["...", "...", "..."]  # list of 100 candidates
ranks = model.rank(query, passages)
for item in ranks[:5]:
    print(f"Score: {item['score']:.2f}, Passage: {passages[item['corpus_id']]}")
```

SBERT provides `model.predict()` for batched scoring and `model.rank()` for ranking a list of passages [17].

Pre-trained Models

CrossEncoder models are also available via the Hugging Face Hub. These are typically named with a `cross-encoder/` prefix. Examples include:

- **Re-ranking models:** e.g. `cross-encoder/ms-marco-MiniLM-L-6-v2`, `cross-encoder/ms-marco-TinyBERT-L2-v2`.
- **STS models:** e.g. `cross-encoder/stsb-roberta-large`.
- **NLI classification:** e.g. `cross-encoder/nli-deberta-v3-base`.

These models are trained specifically for ranking or classification tasks. When choosing one, use a model trained on the closest task (e.g. an MS MARCO cross-encoder for search relevance, an NLI model for entailment tasks).

Training Overview

Cross-encoders are trained similar to sentence transformers but with cross-input. A **CrossEncoderTrainer** handles training loops. Key points:

- **Model:** Initialized with `CrossEncoder(model_name_or_path, num_labels, ...)` [18]. If `num_labels=1`, it does regression (score 0–1 with Sigmoid); if >1, it's a classifier (Softmax output).
- **Dataset:** Should provide pairs `(text1, text2)` and either a regression label or class label.
- **Loss:** Typical losses are `CrossEntropyLoss` for classification tasks, or `MSELoss` for regression of a target score. SBERT also offers `BinaryCrossEntropyLoss` or ranking losses (e.g. *ListNet*) for reranking.
- **Training Arguments:** As usual (batch size, epochs, warmup).
- **Example:** Fine-tune on QNLI (question answering NLI) or MS MARCO for re-ranking.

Training a CrossEncoder might look like:

```
from sentence_transformers import CrossEncoder, CrossEncoderTrainer, losses

model = CrossEncoder("cross-encoder/bert-base", num_labels=1)
# Prepare a DataLoader or Hugging Face dataset with 'text' and 'label'
loss = losses.MSELoss(model)  # if training to match target scores
trainer = CrossEncoderTrainer(model=model, train_dataloader=train_dataloader,
```

```
loss=loss)
trainer.train()
```

Loss Overview

Cross-encoders use losses appropriate to scoring tasks. Common choices:

- **CrossEntropyLoss:** For (sentence1, sentence2) with categorical label (softmax over classes). Often used for entailment/NLI fine-tuning.
- **BinaryCrossEntropyLoss:** For binary relevance/regression on [0,1] scale.
- **Ranking Losses:** SBERT also provides listwise losses (ListMLE, LambdaLoss) suitable for learning-to-rank scenarios.
- **MultipleNegativesRankingLoss:** Less common for cross-encoders, but SBERT does allow it if you structure data as anchor-positive pairs in a batch.

Since CrossEncoders see both inputs together, you typically use a single combined input (no separate pooling). The loss is applied to the model's output logits or score. For example, training on QNLI uses `CrossEntropyLoss` with 2 classes (entail/not-entail) on premise-hypothesis pairs.

Training Examples

Example tasks include:

- **Semantic Search Re-ranking:** Train on (query, positive passage, negative passages) with a contrastive loss (e.g. *BinaryCrossEntropyLoss* where label=1 for pos, 0 for neg) to refine ranking.
- **Question-Answer Relevance:** Fine-tune on QA pairs with *CrossEntropyLoss* on correctness.
- **General NLI:** Use (premise, hypothesis, label) with *CrossEntropyLoss*.

A snippet for binary classification (e.g. yes/no relevance):

```
from sentence_transformers import CrossEncoder, CrossEncoderTrainer, losses

model = CrossEncoder("cross-encoder/bert-base", num_labels=2)
# DataLoader yields (input_ids, attention_mask, labels)
loss = losses.CrossEntropyLoss(model)  # multiclass (here 2 classes)
trainer = CrossEncoderTrainer(model=model, train_dataloader=train_loader,
loss=loss)
trainer.train()
```

Refer to the *Training Examples* section of the SBERT docs for detailed scripts (e.g. *Natural Language Inference*, *Quora*, *MS MARCO Cross-Encoder*, *Rerankers*).

Sparse Encoder

Usage

Sparse Encoders (e.g. **SPLADE**) produce sparse high-dimensional vectors (often 10k+ dims) meant for retrieval with inverted indices. Usage is similar to SentenceTransformer:

```
from sentence_transformers import SparseEncoder

model = SparseEncoder("naver/splade-cocondenser-ensembles")
text = "This is an example sentence"
sparse_embedding = model.encode(text)  # SciPy sparse vector or dense vector depending on model
```

Sparse embeddings can be used for **semantic search** by computing dot-product (or using FAISS with sparse index). SBERT provides integration with search systems (Elasticsearch, OpenSearch, Qdrant) for sparse vectors.

Pre-trained Models

Pretrained sparse models include:

- **Core SPLADE Models:** e.g. `naver/splade-cocondenser`, `naver/splade-distil`.
- **Inference-free SPLADE Models:** variants optimized for faster inference (smaller vocab or distillation).
- **Model Collections:** e.g. bilingual or ensemble SPLADE models.

Check SBERT's *Sparse Encoder* model page for up-to-date listings. For example, `naver/bert-base-splade-covid` is trained on Covid data. Always match the model to your language and domain.

Training Overview

Training a SparseEncoder is analogous to SentenceTransformer but uses specialized loss and regularization:

- **Model:** Typically consists of a Transformer and a *SparseAutoEncoder* or *SPLADE Pooling* module that produces sparse weights.
- **Dataset:** Usually an information retrieval (IR) dataset (e.g. MS MARCO passages) or any (query, document) pairs.
- **Loss:** SBERT supports *SpladeLoss* (encourages sparsity) and *CSRLoss* (control sparsity vs reconstruction) [19], as well as standard ranking losses (*SparseMultipleNegativesRankingLoss*).
- **Trainer:** `SparseEncoderTrainer` manages training (similar API to others).

During training, a *splade-style* loss encourages a few large activation buckets per vector. For example, *SpladeLoss* adds L1 penalty for sparsity. After training, each text is mapped to a sparse vector of terms (like a TF-IDF weighted bag-of-words).

Dataset Overview

Sparse training typically uses **(query, passage, negatives)** datasets (like MS MARCO or Wikipedia retrieval sets). The "Dataset Overview" principles still apply (see Sentence Transformer section), but often one works with triplets or hard negatives. SBERT's **Hard Negative Mining** utilities can be useful (mine hard negatives from a bi-encoder to train the sparse model).

Loss Overview

Sparse-specific losses include:

- **SpladeLoss:** Encourages sparse term activations (L1 regularization) while learning query-document relevance [19].
- **CSRLoss (Contrastive Sparse Retrieval Loss):** Another sparse retrieval loss.
- **SparseMultipleNegativesRankingLoss:** Extension of MNR for sparse models (treats other in-batch passages as negatives).

SBERT's loss table shows these under *Sparse-specific Loss Functions*. In addition, one can combine *MarginMSELoss* or *DistillKLDivLoss* if doing knowledge distillation for sparse.

Training Examples

Examples include:

- **MS MARCO IR (Bi-Encoder):** Train a SparseEncoder on MS MARCO with a triplet or pairwise loss.
- **Knowledge Distillation:** Use a strong dense bi-encoder as teacher and distill into a sparse model (SparseMarginMSELoss).
- **Multilingual IR:** Train sparse for retrieval in new languages.

For instance, to train a sparse retriever on MS MARCO, you might use `MultipleNegativesRankingLoss` or `SparseMarginMSELoss` with `(query, pos, negs)` triples. The SBERT examples include scripts like *MS MARCO Sparse MNRL* and *Information Retrieval*.

Cross-Lingual Training (Multilingual Fine-Tuning)

SBERT supports extending monolingual models to new languages via **knowledge distillation** on parallel corpora [20]. The idea: a teacher model (e.g. English SBERT) provides embeddings for English sentences; a student model (initialized with a multilingual base like XLM-R) is trained so that translated sentences map to the same embeddings [21]. In practice:

- **Parallel Data:** Use large parallel corpus (e.g. OPUS parallel sentences or the provided "parallel-sentences-talks" datasets) [22]. Each example has `english` and `non_english`.
- **Labels:** Compute teacher embeddings for the English sentence, then train student so that `student(non_english) ≈ teacher(english)` using *MSELoss* [23] [8].
- **Training:** Use `SentenceTransformerTrainer` with `MSELoss(student_model)`, providing the teacher labels.

- **Evaluation:** Monitor *TranslationEvaluator* (accuracy of matching translated pairs) and *EmbeddingSimilarityEvaluator* on cross-lingual STS [24]. You can also compute MSE between student and teacher embeddings as a metric [25].

This approach yields models like `paraphrase-multilingual-MiniLM-L12-v2`. Example code snippet (from docs):

```
from sentence_transformers import SentenceTransformer, MSEEvaluator
from datasets import load_dataset

# Load parallel sentences dataset (en-fr, for example)
dataset = load_dataset("sentence-transformers/parallel-sentences-talks", "en-
fr", split="train")
# Compute teacher embeddings (English) and attach as labels
teacher = SentenceTransformer("all-MiniLM-L6-v2")
dataset = dataset.map(lambda x: {'label': teacher.encode(x['english'])},
batched=True)

# Define student model (XLM-R base)
student = SentenceTransformer("xlm-roberta-base")
loss = losses.MSELoss(student)
trainer = SentenceTransformerTrainer(model=student, train_dataset=dataset,
loss=loss)
trainer.train()
```

During/after training, use `MSEEvaluator` or `TranslationEvaluator` to check cross-lingual alignment [25] [26]. For example, `TranslationEvaluator(english, french, ...)` reports the accuracy of retrieving the correct translation [26].

Evaluation (MTEB Benchmark)

SBERT integrates the **Massive Text Embedding Benchmark (MTEB)** for evaluating embedding models across tasks. To use it:

1. **Install MTEB:**

```
pip install mteb
```

2. **Run evaluations:** You can specify tasks or full benchmarks. For example, evaluate specific STS tasks:

```
import mteb
from sentence_transformers import SentenceTransformer
```

```
model = SentenceTransformer("all-MiniLM-L6-v2")
tasks = mteb.get_tasks(tasks=["STS22.v2"], languages=["eng"])
evaluation = mteb.MTEB(tasks=tasks)
results = evaluation.run(model, output_folder="results/")
```

1. **Use filters:** Choose tasks by type/domain. E.g. medical retrieval tasks in English:

```
tasks = mteb.get_tasks(task_types=["Retrieval"], domains=["Medical"],
languages=["eng"])
results = mteb.MTEB(tasks=tasks).run(model, output_folder="results/")
```

2. **Run full benchmarks:** Use `get_benchmark()`, e.g.:

```
bench = mteb.get_benchmark("MTEB(eng, v2)")
results = mteb.MTEB(tasks=bench).run(model, output_folder="results/")
```

3. **Additional args:** Use `encode_kwargs` to pass `batch_size`, `normalize_embeddings`, etc. MTEB will respect prompts defined in the model config [27].

4. **Results handling:** MTEB caches results. After `run()`, iterate through `results` to inspect scores:

```
for task_res in results:
    print(f"{task_res.task_name}: {task_res.get_score():.4f}")
```

   (Shown in example [28]).

To submit to the MTEB leaderboard, add your model metadata and push results as per [MTEB docs] [25†L660-L669].

Speeding up Inference

SBERT provides options to accelerate embedding and scoring:

- **PyTorch backend optimizations:** Use mixed precision on GPU. For example, initialize with `model = SentenceTransformer(..., model_kwargs={"torch_dtype":"float16"})` or call `model.half()` [1]. This yields ~2x speedup with minimal accuracy loss. Similarly, use `bfloat16` with `model.bfloat16()` for better precision on supporting hardware [29].
- **ONNX Runtime:** You can export models to ONNX for CPU/GPU acceleration. Install SBERT with ONNX extras (`pip install sentence-transformers[onnx-gpu]`). Then:

```
model = SentenceTransformer("all-MiniLM-L6-v2", backend="onnx")
embeddings = model.encode(sentences)
```

SBERT will auto-export to ONNX (or use an existing ONNX if found) [30]. You can configure `model_kwargs={"provider":"CUDAExecutionProvider"}`, `file_name`, etc. After exporting once, save the model (`model.save_pretrained()`) to avoid re-export on reuse [31].

- **ONNX Optimization:** Use `export_optimized_onnx_model(model, optimization_config, save_path)` to apply Hugging Face Optimum optimizations [32]. This can fuse operations for CPU/GPU speed. For example, level `"O3"` yields significant speedups [33].
- **OpenVINO:** Similar support for exporting to OpenVINO (see SBERT docs *Speeding up Inference* for details).
- **Multi-GPU / Multi-Process:** SBERT supports DataParallel and PyTorch's FSDP for training. For encoding, you can use multiple processes. The `model.encode()` method has `num_workers` and `batch_size` arguments; see Multi-Process Encoding for details.

Quantization: For extreme size/speed reduction, SBERT provides embedding quantization (see *Embedding Quantization*). You can quantize embeddings to 1-bit or 8-bit to accelerate retrieval (with a small accuracy trade-off) [34] [35].

Embedding Quantization

SBERT includes utilities to quantize embeddings for storage or retrieval speed. Two main types:

- **Binary Quantization:** Threshold float embeddings at 0 to get 1-bit values. This yields a ~32× reduction in memory and very fast Hamming-distance search [36]. A recommended "re-ranking" trick is to retrieve using binary vectors and then re-score top results with the original float query for ~96% of the original performance [37]. Example usage:

```
from sentence_transformers import SentenceTransformer
from sentence_transformers.quantization import quantize_embeddings

model = SentenceTransformer("mixedbread-ai/mxbai-embed-large-v1")
texts = ["A sentence.", "Another sentence."]
# Option 1: encode with binary quantization
binary_embs = model.encode(texts, precision="binary")
# Option 2: encode normally then quantize
embs = model.encode(texts)
binary_embs = quantize_embeddings(embs, precision="binary")
print(embs.shape, binary_embs.shape)  # e.g. (2,1024) vs (2,128) bytes
```

Here `precision="binary"` packs bits into uint8 arrays [38] [39].

- **Scalar (int8) Quantization:** Maps float32 ranges into 256 buckets. Requires a calibration set or known min/max per dimension [40]. Usage:

```python
from sentence_transformers import SentenceTransformer, quantize_embeddings
from datasets import load_dataset

model = SentenceTransformer("mixedbread-ai/mxbai-embed-large-v1")
# Prepare calibration embeddings (e.g. encode a sample corpus)
corpus = load_dataset("nq_open", split="train[:1000]")["question"]
cal_embs = model.encode(corpus)
# Encode queries and quantize
query_embs = model.encode(["I am driving to the lake."])
int8_embs = quantize_embeddings(query_embs, precision="int8",
calibration_embeddings=cal_embs)
```

Quantized embeddings reduce memory by ~4× (32-bit → 8-bit) at modest accuracy loss. Always use ample calibration data to set stable min/max buckets [41].

After quantization, you can use the embeddings with sparse/dense retrieval (e.g. store them in a vectordb). SBERT also provides `quantize_embeddings()` utility as shown.

Package Reference

Key SBERT classes and utilities:

- **SentenceTransformer:** Bi-encoder model class (`sentence_transformers.SentenceTransformer`). Methods: `.encode()`, `.encode_async()`, `.simulate_embedding()`, `.similarity()`, etc. Also `.train()` for compatibility or use `SentenceTransformerTrainer`.
- **SentenceTransformerTrainer:** Training loop for bi-encoders. Arguments include `model`, `train_dataset`, `loss`, etc.
- **CrossEncoder:** Cross-encoder class (`sentence_transformers.CrossEncoder`) [42]. Methods: `.predict(pairs)`, `.rank(query, passages)`, `.train()` (via `CrossEncoderTrainer`).
- **CrossEncoderTrainer:** Trainer for cross-encoders.
- **SparseEncoder:** Sparse encoder class (`sentence_transformers.SparseEncoder`).
- **SparseEncoderTrainer:** Trainer for sparse models.
- **Util (`sentence_transformers.util`):** Helper functions such as:
- `semantic_search()`: run bi-encoder search.
- `paraphrase_mining()`: find paraphrase pairs in a corpus.
- `normalize_embeddings()`, `cos_sim()`, etc.
- `mine_hard_negatives()`: generate hard negatives given a retriever model.
- `export_dynamic_quantized_onnx_model()`, `export_static_quantized_openvino_model()`: for model export/quantization.

A fuller reference of all classes, their methods, and parameters is in the SBERT documentation's *Package Reference* section [43] [44]. For example, the `SentenceTransformer` initializer can take multiple arguments (model name, `device`, etc.), and has an accompanying `SimilarityFunction` enum. The CrossEncoder's `__init__` parameters are documented (e.g. `num_labels`, `activation_fn`, `backend`) [45].

Always consult the docstring or online reference for details on parameters (e.g. `max_length`, `batch_size`) for each class.

Best Practices & Pitfalls

- **Data/Loss Alignment:** Ensure your dataset format matches the loss (e.g. don't feed single sentences to a triplet loss). The SBERT docs stress verifying compatibility [10] [12].
- **Hard Negatives:** Using hard negatives (from e.g. `util.mine_hard_negatives()`) often boosts performance in retrieval tasks [11].
- **Batch Size:** For ranking losses, larger batch sizes yield more in-batch negatives (e.g. MNRLoss with batch of 64 compares 64×64 pairs). Use `CachedMultipleNegativesRankingLoss` if GPU memory is limited. [46]
- **Device and Precision:** When using GPUs, float16/bfloat16 can speed up encoding [1]. But test for accuracy loss. For ONNX, always export once and cache the model to avoid repeated conversion.
- **Evaluation:** Always evaluate on held-out sets. Use MTEB for broad benchmark. For search tasks, use appropriate retrieval metrics (e.g. `InformationRetrievalEvaluator`).
- **Distributed Training:** SBERT supports PyTorch FSDP and multi-GPU training. This is important for large models or datasets, but may require chunking gradients. See *Distributed Training* docs.

References

All information above is drawn from the Sentence Transformers documentation and related sources [2] [47] [12] [48] [1] [49] [42] [17], reflecting the latest SBERT v5+ features and best practices. For code examples and deeper reading, consult the official SBERT docs and GitHub.

---

[1] [29] [30] [31] [32] [33] Speeding up Inference — Sentence Transformers documentation
https://sbert.net/docs/sentence_transformer/usage/efficiency.html

[2] [3] [5] Training Overview — Sentence Transformers documentation
https://sbert.net/docs/sentence_transformer/training_overview.html

[4] [9] [10] [11] [47] Dataset Overview — Sentence Transformers documentation
https://sbert.net/docs/sentence_transformer/dataset_overview.html

[6] [7] [12] [13] [14] [15] [16] [19] [43] [44] [46] Loss Overview — Sentence Transformers documentation
https://sbert.net/docs/sentence_transformer/loss_overview.html

[8] Losses — Sentence Transformers documentation
https://sbert.net/docs/package_reference/sentence_transformer/losses.html

[17] cross-encoder (Sentence Transformers - Cross-Encoders)
https://huggingface.co/cross-encoder

[18] [42] [45] CrossEncoder — Sentence Transformers documentation
https://sbert.net/docs/package_reference/cross_encoder/cross_encoder.html

[20] [21] [22] [23] [24] [25] [26] Multilingual Models — Sentence Transformers documentation
https://sbert.net/examples/sentence_transformer/training/multilingual/README.html

27 28 48 Evaluation with MTEB — Sentence Transformers documentation
https://sbert.net/docs/sentence_transformer/usage/mteb_evaluation.html

34 35 36 37 38 39 40 41 49 Embedding Quantization — Sentence Transformers documentation
https://sbert.net/examples/sentence_transformer/applications/embedding-quantization/README.html